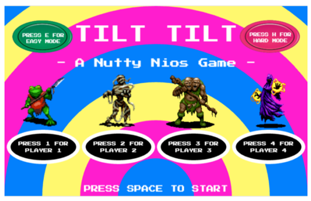
Nutty Nios

dgs119 wh1618 jfe20 ddl20 bpr20

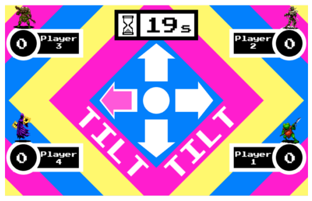
1 Introduction

1.1 Overview

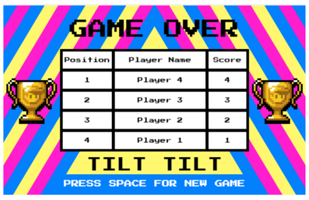
Nutty Nios is a real-time multiplayer browser game based on Konami's Dance Dance Revolution [1]. The game involves players tilting their FPGA to correspond with arrows shown on the browser, and can support up to four players playing at once.



(a) Starting Screen



(b) Gameplay



Position	Player Name	Score
1	Player 4	4
2	Player 3	3
3	Player 2	2
4	Player 1	1

(c) End Screen

Figure 1: Nutty Nios' Game Play

During development, design decisions were made to ensure Nutty Nios was **responsive**, **portable** and **secure**. To meet these requirements, Nutty Nios was evaluated on its node's resource utilization as well as node to gateway and gateway to server round trip times.

1.2 Functional Requirements

The following features of Nutty Nios fulfills the minimum functional requirements of the system:

- Local processing of Accelerometer Data: The FPGA performs fixed point FIR filtering on data from its accelerometer sensor to smoothen it.
- Communicating Information from Node to Server: The FPGA's accelerometer data is processed by a gateway (local computer) to compute the tilt direction of the board. Directions are then published to a secure MQTT broker hosted on AWS.
- Establishing a Cloud Server to Process Events: A Colyseus Server hosted on AWS computes the next state of the game depending on directions published to the MQTT broker.
- Communicating Information from the Server back to the Nodes: The FPGA toggles between sets of filter coefficients depending on the mode of the game - easy or hard - chosen by a player.

2 System Architecture

2.1 System Diagram

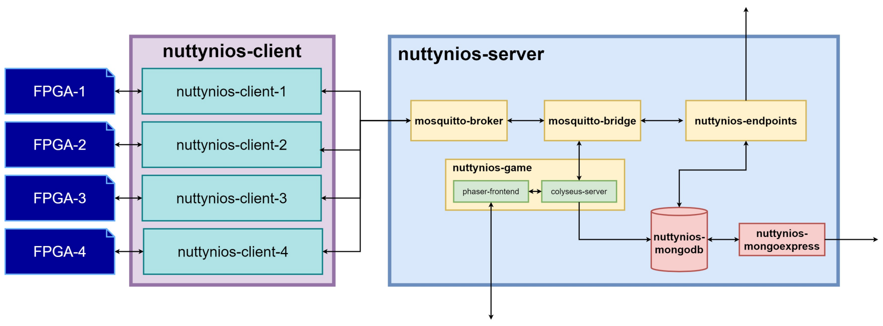


Figure 2: System Architecture of Nutty Nios

2.2 Components

1. Node: Data from the FPGA's accelerometer sensor is filtered and sent to a client connected to it.

2. Client: The client continuously reads the stream of data sent by the board. It then computes the tilt direction of the board and publishes it to the server.

3. Server: A Docker Swarm that listens to the directions published by each gateway and computes the next state of the game. Its composed of the following services:

   (a) MQTT Broker: A server that receives and distributes messages to authenticated clients over MQTT protocol.

   (b) MQTT Bridge: It is connected to the MQTT broker and allows unrestricted access to it for services within the Docker Swarm.

   (c) Colyseus Server: It computes the next state of the game depending on direction messages published to the MQTT broker. It also handles multiple client connections to the game.

   (d) Phaser Frontend: This service listens to the Colyseus server for changes in the game's state and updates the frontend accordingly.

   (e) MongoDB: A no SQL database that stores the leaderboard after each game.

   (f) FastAPI Service: This service exposes endpoints to update and fetch records stored in the database.

3 Node

The main functionality of the node or FPGA was to act as a controller for our Nutty Nios game, but later on many other features were also added to the firmware of the FPGA, such as: programmable led's from the gateway, side scrolling 7-segment display for custom text, and two on-board filter mode choices.

3.1 Architecture

The instantiated IPs in our system include: Nios II/e Softcore Processor, JTAG UART, On-chip Memory, Accelerometer SPI, LED PIO, 6 HEX (7SEG) PIO, Switch PIO, Buttons PIO, and Interval Timer. The resource usage report on the DE-10 Lite board is in the table below.

Compilation Report	
Device	10M50DAF484C7G
Total Logic Elements	2,656/49,760 (5%)
Total Memory Bits	731,264/1,677,312 (44%)

Table 1: Quartus Compilation Report

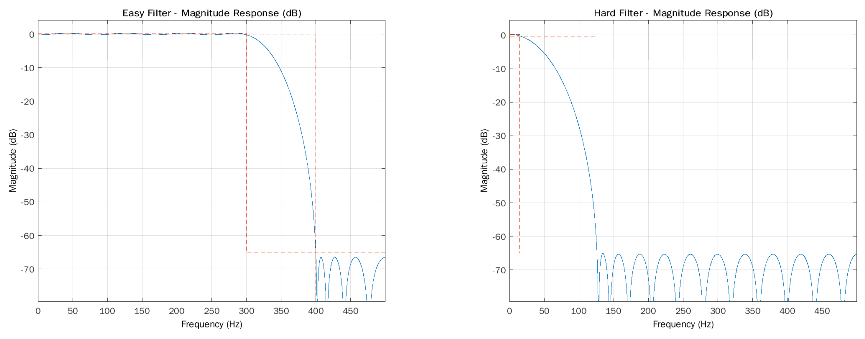
While designing the system as well as the firmware for the system, our team prioritised reducing the number of logical components as well as memory components so that the system could be as portable to devices with lower specifications. This is evident in compilation report as the design had only used 5% of the logical elements and 44% of the memory bits (see Table 1) on the DE-10 Lite FPGA development board. Furthermore, all of the firmware which had been developed could fit onto the on-chip memory of the FPGA, and no additional SDRAM was required.

3.2 Firmware

The firmware of the board handles all the inputs and outputs in a perpetual loop. The details of how they are processed are listed below:

- **Accelerometer Data Output:** The accelerometer data is sampled at a rate of around 200Hz which is controlled by an interval timer set with a particular timeout. Every time the data is sampled, it will also be filtered with the on-board fixed-point FIR filter. Lastly, depending on the difficulty mode of the game, te accelerometer values will then be scaled to make the controller feel more or less sensitive to the user.
- **Text Input:** The FPGA will always listen for messages on the UART port of the board for text input via a non-blocking read from the `stdin` stream. After processing and reading the message, the board will then change its state accordingly.

3.3 FIR Filtering



(a) Easy Filter (b) Hard Filter

Figure 3: Onboard FIR Filters

The magnitude responses of the respective easy and hard filters can be seen in Figure 3. The hard filter has a much lower gain for high frequency inputs. For the same value to be registered by the FPGA, the user must now move the FPGA in a more exaggerated manner. To switch between the filter used, the FPGA listens to commands from the gateway.

3.4 Performance Testing

As the main functionality of the FPGA was to act as a controller, the focus was to design it to be responsive. Therefore the two performance metrics which mattered were the data sampling rate and the latency. Figure 4 in Section 5.4 summarises the performance testing conducted from the node to gateway. With a sampling rate of 185.3 $samples/s$ and round trip time of 3.51 ms, the team concluded the controller was responsive enough for our application.

4 Gateway

4.1 Functionality

The gateway processes accelerometer coordinates written by the FPGA to `nios2-terminal` into a direction movement that is sent over to the server.

Processing is done through simple thresholding and buffering. Thresholding was first performed to determine the direction the board was moved in each sample. These directions were then added to a circular buffer. The board was deemed to be moved in a particular direction, if all directions in the buffer of samples were the same. This is finally published to the MQTT broker hosted in the cloud.

Here, fixed circular buffers were used due to their fast, constant time append and pop operations [2]. They were also sufficiently sized to give the best compromise between delay and smoothing performance. Large buffers smoothened directions well but introduced delays proportional to its size.

Local processing was done at this stage to reduce the load on the board and server. This preserved the rate at which the board was publishing data to the client, and reduced processing required by the server.

4.2 Performance Testing

The data throughput of the gateway was 26.1 $samples/s$; this was significantly lower than the throughput of the node which was 185.3 $samples/s$. The gateway introduces an additional processing delay of $32ms$. However, this is still lower than than trip times taken for packets to be sent to the server - $100ms$ (see Figure 4).

5 Server

The server was implemented using docker and hosted on an AWS instance. Docker was chosen to provide our project with portability and to reap the benefits of containerisation - minimized overhead and reduced development environment conflicts [3]. The server performs three main tasks - networking, hosting the game and managing the database.

5.1 Networking

To support two-way communication between the server and the local nodes, a MQTT broker is hosted on the server. The MQTT protocol was chosen for our project as it is a lightweight and flexible network protocol [4] that can be implemented on heavily constrained device hardware as well as high latency networks.

Our MQTT broker is configured such that only clients with valid client certificates will be able to establish a connection with the broker. On top of improved security, it also helps ensure reliable gameplay by preventing unauthenticated clients from flooding the broker. For unrestricted networking within the docker swarm, a MQTT bridge is connected to the MQTT broker and hosted on the server as well.

5.2 Game

A web browser based game was chosen to offer greater accessibility and versatility of our project. Phaser [5] was used to render the frontend of the game while Colyseus was used to provide support for multiplayer games. The game was deliberately designed such that processing at the Phaser frontend was kept to a minimum so that the web application can be fast and responsive for all players.

Colyseus is a server authoritative framework [6] that provides synchronization of states between all connected clients. Within the Colyseus service, a connection to the MQTT bridge is established to listen to direction inputs from the gateway. The Colyseus service keeps track of all direction inputs, scores of all players, as well as the player rankings. Messages are then sent to the Phaser frontend where changes will be rendered accordingly.

Our game offers two different difficulty options. Players can press the key 'E' for easy or 'H' for hard. This triggers a message to be sent from Phaser frontend to the Colyseus service. For communication from the server to the node, the Colyseus service publishes messages to the MQTT bridge which the gateway subscribes to through the broker. Upon receiving a message, the gateway simply sends a command to the FPGA to change the filter used on its accelerometer data.

The same flow of data is used to display the individual player scores on their specific FPGA at the end of each game.

5.3 Database

MongoDB - a NoSQL database - was used to store player scores at the end of each game. A NoSQL databse was chosen as it is fast to develop, can store unstructured data, and its schemas can be updated quickly and easily [7]. Stored documents contained a unique game id as its primary key, a player id, and score.

To communicate with the database, a FastAPI service was instantiated that exposed endpoints to GET and POST documents to the database.

5.4 Performance Testing

The RTT for the transfer of packets between the gateway to the server was measured to be $197.8ms$ (see Figure 4). This is the bottleneck of the system as it is significantly longer than the node to the gateway trip time ($1.75ms$) and the processing delay in the gateway ($32ms$). It is also worth noting that limited processing is done by the gateway on the messages received from the server before an appropriate command is sent to the node.

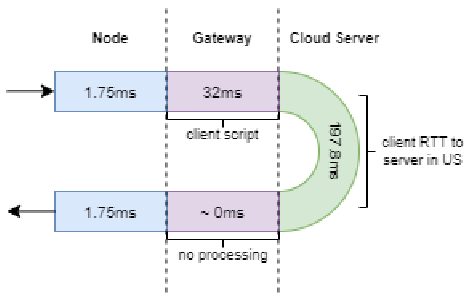


Figure 4: System RTT of Nutty Nios

This latency can be attributed to the large geographical distance from the gateway to our AWS EC2 instance that was hosted in the US. Hosting the AWS server within the UK could mitigate this bottleneck.

6 Testing

The hardware, client and server components were split into different submodules, and development was done on separate branches of each repository (see Figure 5). Separate repositories containerized bugs, and feature branches avoided introducing bugs into functional prototypes.

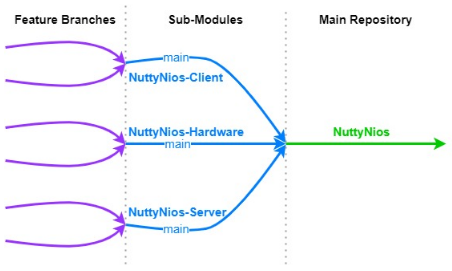


Figure 5: Development Flow

Feature branches were merged only after being tested at each submodule and upon merges, the entire system was tested for functionality (see Figure 6). Any bugs found at system level tests repeated the testing flow. This iterative testing approach allowed us to catch bugs faster.

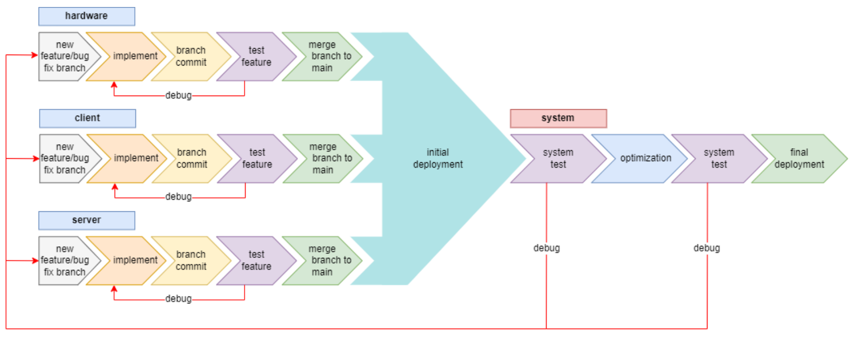


Figure 6: Testing Approach [8]

References

[1] Dance dance revolution. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Dance_Dance_Revolution

[2] Deque objects. Python. [Online]. Available: https://docs.python.org/3/library/collections.html#collections.deque

[3] Containerisation. IBM. [Online]. Available: https://www.ibm.com/uk-en/cloud/learn/containerization

[4] M. Yuan. Mqtt protocol. IBM. [Online]. Available: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/

[5] Phaser. Phaser. [Online]. Available: https://phaser.io/

[6] Colyseus. Colyseus. [Online]. Available: https://www.colyseus.io/

[7] Nosql database. MongoDB. [Online]. Available: https://www.mongodb.com/nosql-explained/advantages

[8] J. Mendes. Testing approach. GitHub. [Online]. Available: https://github.com/JosiahMendes/MIPS32-T501